The LLVM Compiler Infrastructure

Novel Capabilities, Current Status, Future Direction

Chris Lattner

lattner@cs.uiuc.edu

aka t-chrisl

Joint work with:

Vikram Adve

vadve@cs.uiuc.edu

http://llvm.cs.uiuc.edu/

Microsoft Research: June 30, 2004

The LLVM Compiler Infrastructure

Novel Capabilities, Current Status, Future Direction

Chris Lattner

lattner@cs.uiuc.edu

aka t-chrisl

Joint work with:

Vikram Adve

vadve@cs.uiuc.edu

http://llvm.cs.uiuc.edu/

Microsoft Research: June 30, 2004

Spoiler for the rest of this talk

- **LLVM code representation is quite unique:**
  - IR is not "lowered" through the compiler
  - Truly source-language/target independent
- **LLVM supports very high-level opt/analysis:**
  - Analysis and restructuring of code
  - Supports most classical optimizations as well
- **LLVM supports very low-level work as well:**
  - Novel microprocessor/micro-architectural designs
  - Requires high-level info and low-level representation

Talk Outline

- **Lifelong Program Analysis & Optimization**
  - **LLVM Virtual Instruction Set**
  - **LLVM Compiler Architecture**
- **LLVM as a Compiler Infrastructure**
- **Recursive DS Analysis / Transformations**
- **Virtual Instruction Set Computing**
- **Summary**

Life-Long Program Optimization:

- **Multiple-stages of analysis & transformation:**
  - compile-time, link-time, install-time, run-time, idle-time
  - Use aggressive interprocedural optimizations
  - Gather and exploit end-user profile information
  - Tune the application to the user's hardware
- **But what constraints do we have to meet?**
  - Can't interfere with the build process!
  - Must support multiple source-languages!
  - Must integrate with legacy systems and components!

"LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation"
Chris Lattner and Vikram Adve. CGO'2004

Five key capabilities are needed:

1. **A persistent, rich code representation**
   - Enables analysis & optimization throughout lifetime
2. **Offline native code generation**
   - Must be able to generate high-quality code statically
3. **Profiling & optimization in the field**
   - Adapt to the **end-user's** usage patterns
4. **Language independence**
   - No runtime, object model, or exception semantics
5. **Uniform whole-program optimization**
   - Allow optimization across languages and runtime

What about previous approaches?

Approach	Persistent Rich Code	Offline Codegen	End-user Profiling	Transparent runtime	Uniform whole-prog.
Source-level Compilers		✓		✓	
Source-level Link-time IPO	through link-time	✓		✓	
Machine Code Optimizers	?	✓	✓	✓	✓
High-Level Virtual Machines	✓		✓		user code only
LLVM	✓	✓	✓	✓	✓

Our approach: The LLVM System

- **Use a low-level, but typed, representation:**
  - Type information allows important high-level analysis
  - Code representation is truly language neutral
  - Allow off-line and runtime native code generation

- **Our specific contributions:**
  - **Novel features for language independence:**
    - Typed pointer arithmetic, exception mechanisms
  - **Novel capabilities:**
    - First to support all 5 capabilities for lifelong optzn

Why not a HLL VM like CLI/JVM?

- **Differing goals ⇒ differing representations:**
  - HLL VMs: classes, inheritance, mem. mgmt, runtime…
  - LLVM: calls, load/stores, arithmetic, addressing, etc…
- **Implications:**
  - *Verifiable* CLI is not truly language neutral
  - Cannot optimize VM code **into** the application code
  - HLL VMs require specific runtime environments
  - CLI is not used directly for compiler optzn/analysis
- **LLVM complements high-level VMs:**
  - HLL VM can be implemented in terms of LLVM!

Talk Outline

- **Lifelong Program Analysis & Optimization**
  - **LLVM Virtual Instruction Set**
  - **LLVM Compiler Architecture**
- **LLVM as a Compiler Infrastructure**
- **Recursive DS Analysis / Transformations**
- **Virtual Instruction Set Computing**
- **Summary**

Talk Outline

- Lifelong Program Analysis & Optimization
  - LLVM Virtual Instruction Set
  - LLVM Compiler Architecture
- LLVM as a Compiler Infrastructure
- Recursive DS Analysis / Transformations
- Virtual Instruction Set Computing
- Summary

LLVM Instruction Set Overview #1

- **Low-level and target-independent semantics**
  - RISC-like three address code
  - Infinite virtual register set in SSA form
  - Simple, low-level control flow constructs
  - Load/store instructions with typed-pointers

```
for (i = 0; i < N;
     ++i)
  Sum(&A[i], &P);
```

```
loop.
  %i.1 = phi int [ 0, %bb0 ], [ %i 2, %loop ]
  %AiAddr = getelementptr float* %A, int %i 1
  call void %Sum(float %AiAddr, %pair* %P)
  %i 2 = add int %i 1, 1
  %tmp 4 = setlt int %i 1, %N
  br bool %tmp.4, label %loop, label %outloop
```

LLVM Instruction Set Overview #2

- **High-level information exposed in the code**
  - Explicit dataflow through SSA form
  - Explicit control-flow graph (even for exceptions)
  - Explicit language-independent type-information
    - Explicit typed pointer arithmetic

```
for (i = 0, i < N;
     ++i)
   Sum(&A[i]  &P),
```

```
loop.
  %i.1 = phi int [ 0, %bb0 ], [ %i 2, %loop ]
  %AiAddr = getelementptr float* %A, int %i.1
  call void %Sum(float %AiAddr, %pair* %P)
  %i 2 = add int %i 1, 1
  %tmp 4 = setlt int %i 1, %N
  br bool %tmp.4, label %loop, label %outloop
```


Chris Lattner lattner@cs.uiuc.edu

LLVM Type System Details

- **The entire type system consists of:**
  - Primitives: void, bool, float, ushort, opaque, ...
  - Derived: pointer, array, structure, function
  - No high-level types: type-system is language neutral!
- **Source language types are lowered:**
  - e.g. `T&` → `T*`
  - e.g. `class T { int X; }` → `{ S, int }`
- **Type system allows arbitrary casts:**
  - Allows expressing non-type-safe languages, like C
  - Does not provide safety or verifiability

LLVM Instruction Set Overview #2

- **High-level information exposed in the code**
  - Explicit dataflow through SSA form
  - Explicit control-flow graph (even for exceptions)
  - Explicit language-independent type-information
    - Explicit typed pointer arithmetic

```
for (i = 0, i < N;
     ++i)
  Sum(&A[i]  &P),
```

```
loop
  %i.1 = phi int [ 0, %bb0 ], [ %i 2, %loop ]
  %AiAddr = getelementptr float* %A, int %i 1
  call void %Sum(float %AiAddr, %pair* %P)
  %i 2 = add int %i 1, 1
  %tmp 4 = setit int %i 1, %N
  br bool %tmp 4, label %loop, label %outloop
```

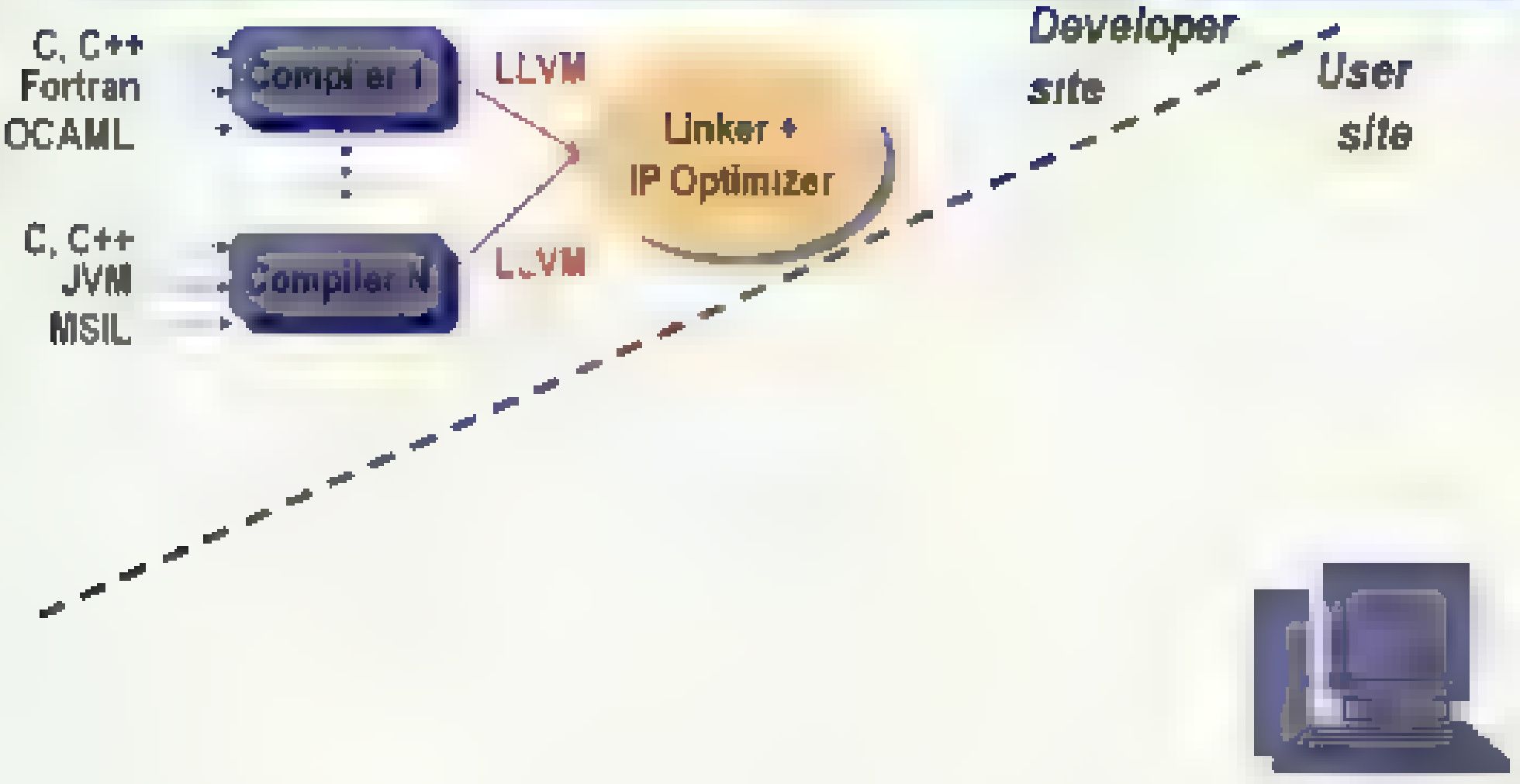
LLVM Type System Details

- **The entire type system consists of:**
  - Primitives: void, bool, float, ushort, opaque, ...
  - Derived: pointer, array, structure, function
  - No high-level types: type-system is language neutral!
- **Source language types are lowered:**
  - e.g. `T[]` → `T*`
  - e.g. `class T { S { int X, }` → `{ S, int }`
- **Type system allows arbitrary casts:**
  - Allows expressing non-type-safe languages, like C
  - Does not provide safety or verifiability

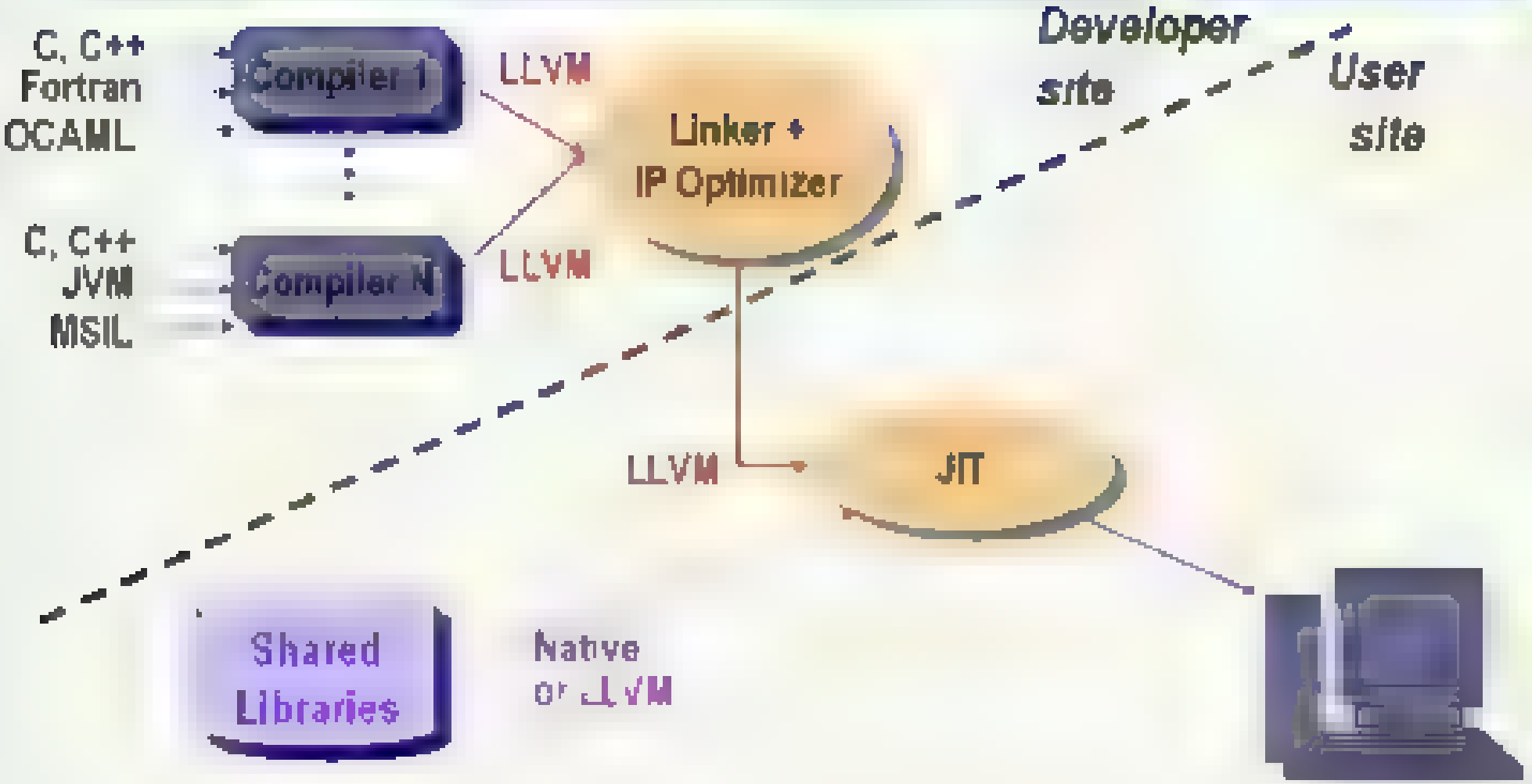
Talk Outline

- Lifelong Program Analysis & Optimization
  - LLVM Virtual Instruction Set
  - LLVM Compiler Architecture
- LLVM as a Compiler Infrastructure
- Recursive DS Analysis / Transformations
- Virtual Instruction Set Computing
- Summary

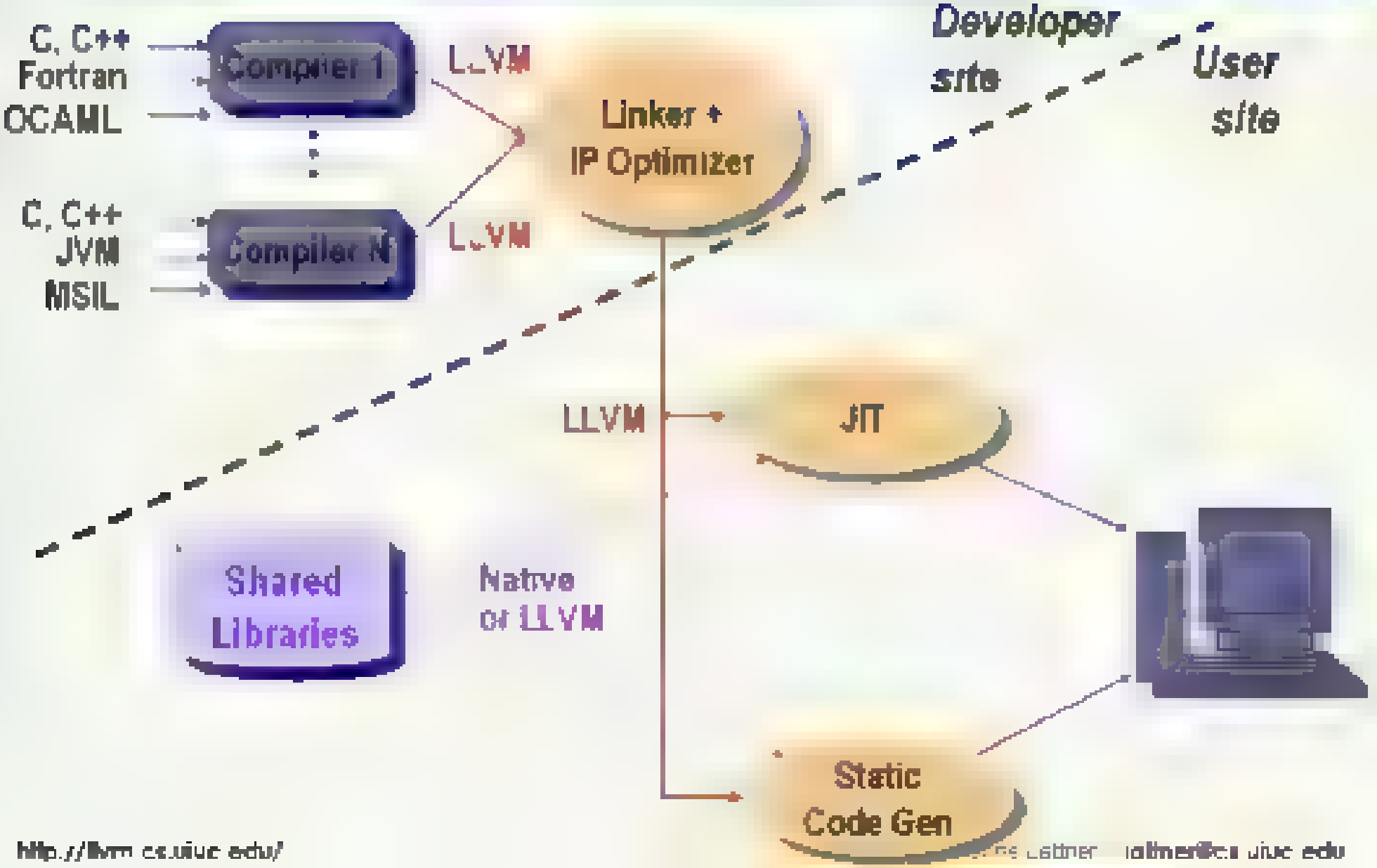
LLVM Compiler Architecture



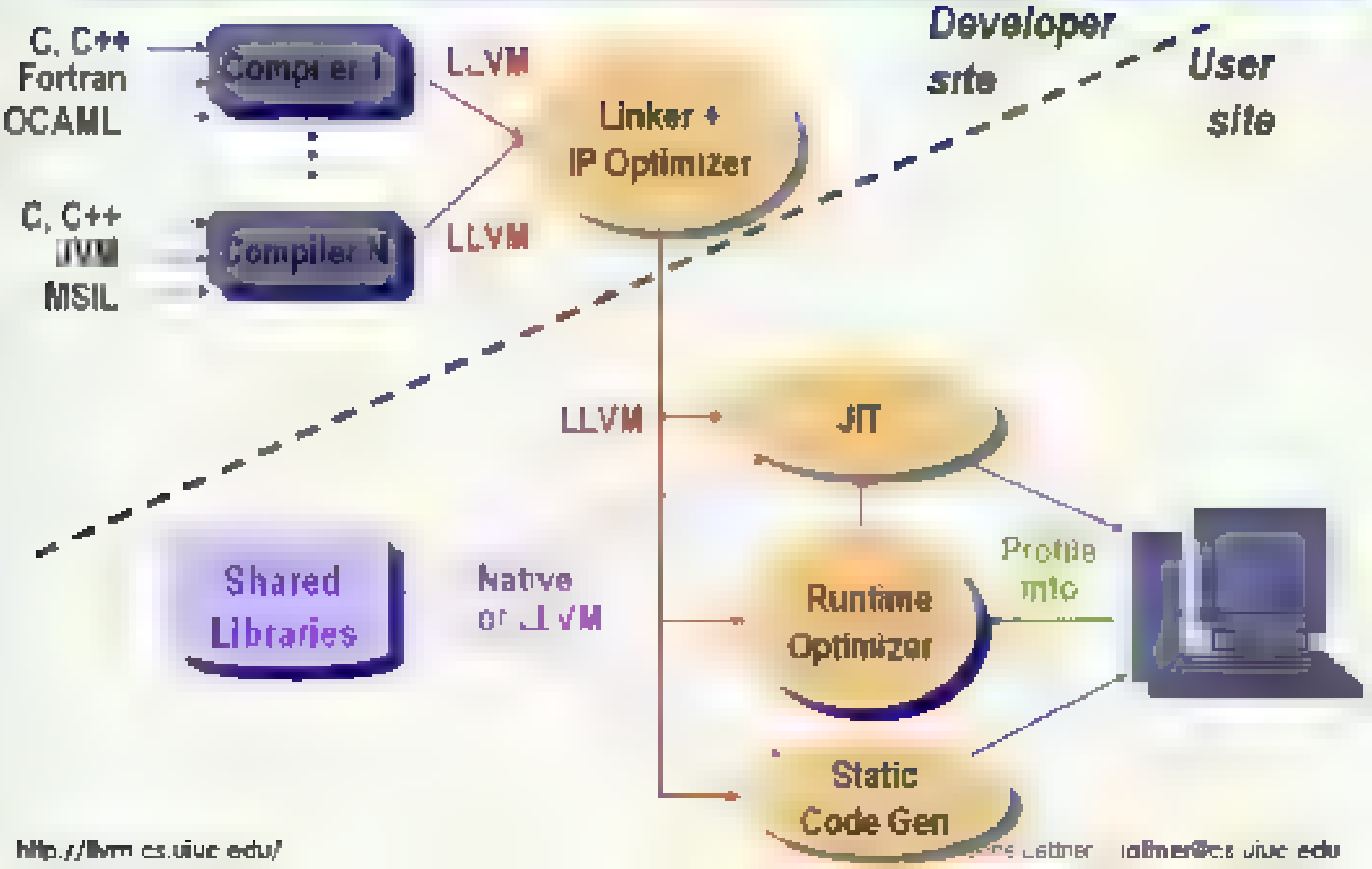
LLVM Compiler Architecture



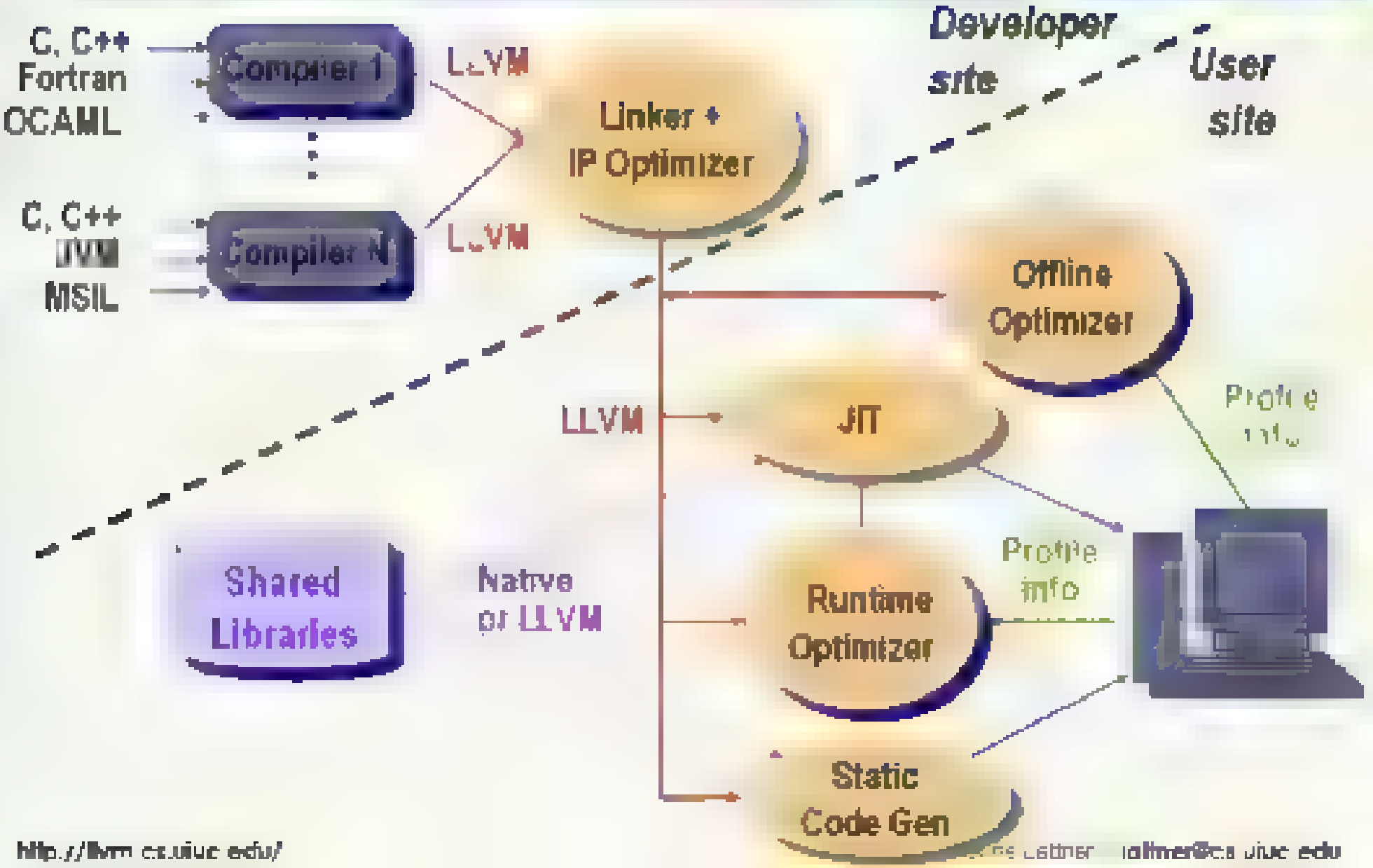
LLVM Compiler Architecture



LLVM Compiler Architecture



LLVM Compiler Architecture



LLVM provides all five capabilities:

1. **A persistent, rich code representation:**
   - LLVM to LLVM optimizations can happen at any time
2. **Offline native code generation:**
   - Generate high-quality machine code, retaining LLVM
3. **Profiling & optimization in the field:**
   - Runtime and offline profile-driven optimizers
4. **Language independence:**
   - Low-level inst set & types with transparent runtime
5. **Uniform whole-program optimization:**
   - Optimize across source-language boundaries

Talk Outline

- Lifelong Program Analysis & Optimization
- LLVM as a Compiler Infrastructure
  - Bugpoint automated bug finder
- Recursive DS Analysis / Transformations
- Virtual Instruction Set Computing
- Summary

A Compiler Infrastructure?

- **Composable *libs* for building compiler *tools*:**
  - Language front-ends, static/JIT code generators, aggressive optimizations, static analysis systems, profile feedback, target information, .
- **Easy to write and debug compiler parts:**
  - Many samples, automated debugging tools, modularity
- **Encourage reuse, but don't require it:**
  - Can write a new register allocator or optimization
  - Can completely replace the code generator

Primitive LLVM Tools

- **'llvm-as', 'llvm-dis': translate .ll ↔ .bc**
- **'llvm-link': Link two LLVM files together**
- **'opt': Run pass sequence on an LLVM file**
  - Useful for unit testing, manual experimentation, etc
  - opt –licm –gcse x.bc –o y.bc
- **'llc': LLVM static code generator**
  - Codegen llvm to a textfile: llc x.bc –march=x86 –o x.s
- **'lli': LLVM Execution Engine (JIT/Interpreter)**
  - lli ls.bc -l

Tools used by the C/C++ compiler

- **'gccas': Compile-time optimizer:**
  - Parse .ll file
  - Run a canned sequence of 41 LLVM passes
  - Output a .bc file
- **'gccld': Linker and link-time optimizer:**
  - Links .bc files, libraries, handles searching .a files
  - Runs 20 more LLVM passes
  - Can output .bc file, or native exe using 'llc'
- **Tools are built by picking the appropriate components and writing glue code**

The Bugpoint automated bug finder

- **Simple idea: automate 'binary' search for bug**
  - Bug isolation which passes interact to produce bug
  - Test case reduction reduce input program
- **Optimizer/Codegen crashes:**
  - Throw portion of test case away, check for crash
    - If so, keep going
    - Otherwise, revert and try something else
  - Extremely effective in practice
- **Simple greedy algorithms for test reduction**
- **Completely black-box approach**

Debugging Miscompilations

- **Optimizer miscompilation:**
  - Split testcase in two, optimize one. Still broken?
  - Keep shrinking the portion being optimized
- **Codegen miscompilation:**
  - Split testcase in two, compile one with CBE, broken?
  - Shrink portion being compiled with non CBE codegen
- **Code splitting granularities:**
  - Take out whole functions
  - Take out loop nests
  - Take out individual basic blocks

The Bugpoint automated bug finder

- **Simple idea: automate 'binary' search for bug**
  - Bug isolation which passes interact to produce bug
  - Test case reduction reduce input program
- **Optimizer/Codegen crashes:**
  - Throw portion of test case away, check for crash
    - If so, keep going
    - Otherwise, revert and try something else
  - Extremely effective in practice
- **Simple greedy algorithms for test reduction**
- **Completely black-box approach**

Debugging Miscompilations

- **Optimizer miscompilation:**
  - Split testcase in two, optimize one. Still broken?
  - Keep shrinking the portion being optimized
- **Codegen miscompilation:**
  - Split testcase in two, compile one with CBE, broken?
  - Shrink portion being compiled with non CBE codegen
- **Code splitting granularities:**
  - Take out whole functions
  - Take out loop nests
  - Take out individual basic blocks

How well does this thing work?

- **Extremely effective:**
  - Can often reduce a 100K LOC program and 60 passes to a few basic blocks and 1 pass in 5 minutes
  - Crashes are found much faster than miscompilations
    - no need to run the program to test

- **Limitations:**
  - Program must be deterministic
    - or modified to be so
  - Finds "a" bug, not "the" bug

LLVM Compiler Status

Public releases in Oct 03, Dec 03, Mar 04

Many downloads, external contributors, ~200K LOC

http://llvm.cs.uiuc.edu/

- **Release includes:**
  - **Front ends:** C, C++ (based on GCC parser)
  - **Back ends:** C, Sparc, X86 (offline or online), [PPC soon]
  - **EE System:** Sparc JIT, X86 JIT, interpreter
  - **Link-time IPO, many global opts, profile feedback, aggressive alias analysis ...**
- **Under development:**
  - **JVM, MSIL, OCAML** front-ends (plus Python, Ruby, Scheme)
  - **Trace-driven runtime optimizer**
  - **Many other derivative projects**

Recursive Data Structures

- **RDS are widely used by many programs:**
  - Linked-lists, binary trees, graphs, heaps, …
- **RDS perform poorly on modern architectures:**
  - RDS nodes are spread randomly in memory
  - Irregular access patterns are bad for locality
  - Small pointer chasing loops are common

Standard Approaches

- **Analyses and xforms focus on primitives:**
  - Disambiguate/eliminate individual loads and stores
  - Reorder, split, or merge structure definitions
  - Heuristics for co-locating heap objects
- **Q: Can we target *entire data structures*?**

Why haven't we done this yet?

- **Existing analyses are usually conservative:**
  - Treat all objects allocated by one malloc the same
  - Type-safe language required for field-sensitivity
    - Type-unsafe languages implies extremely expensive or unsound analyses
- **Aggressive analyses are very expensive**
  - e g shape analysis
  - Usually can't work without **whole** program
- **Runtime layout of heap objects is unknown:**
  - Limits approaches to the simple techniques earlier

The *Macroscopic* Approach

Analyze and Transform Entire Data Structures

- **Data Structure Analysis:**
  - An *aggressive* & *scalable* analysis **for the real world**
  - Identifies data structures & their properties
  - Context-sensitive field-sensitive, flow-insensitive
- **Automatic Pool Allocation:**
  - Transform program to allocate from **memory pools**
  - Provides partial layout control of nodes to the compiler
  - Can identify dynamic DS objects at ***runtime***
- **Many applications of the above are possible**

Talk Outline

- Lifelong Program Analysis & Optimization
- LLVM as a Compiler Infrastructure
- Recursive DS Analysis / Transformations
  - Data structure analysis
  - Automatic pool allocation
- Virtual Instruction Set Computing
- Summary

Data Structure Analysis: *Properties*

Context-sensitive, field-sensitive, yet scalable

Names heap objects by full acyclic call paths

- **2 Compromises:**
  - (a) unification-based (b) flow-insensitive
- **Scalable and very fast:**
  - ~ O(n log n): 137K lines of C code in 8 seconds
  - Field-sensitive only for nodes with unique type
  - Almost no iteration at all
- **Designed for the real world (aka harsh realities of C):**
  - Incomplete programs type-unsafe code
  - Function pointers and recursion
  - Varargs setjmp/longjmp EH
  - Does not need a call graph provided

Data Structure Analysis: *What's New?*

- **Fine grained tracking of incomplete information:**
  - Handle incomplete programs, partially resolved function pointers
  - Speculative field sensitivity
- **Walk SCCs of call-graph with incremental call graph**
  - ***Incremental*** ⇒ **discover call graph during analysis**
  - ***Walk SCCs*** ⇒ **non-iterative algorithm**
- **Field-sensitive *only* for type-safe memory objects**
- (Not new) **Context-sensitivity *can be scalable* in a unification based algorithm**

Important aspects of DSA

Field sensitivity + full context sensitivity

Identifies *instances* and *connectivity* of RDS

DSA also captures important properties of memory objects

```
int G;

void twoLists() {
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
}
```

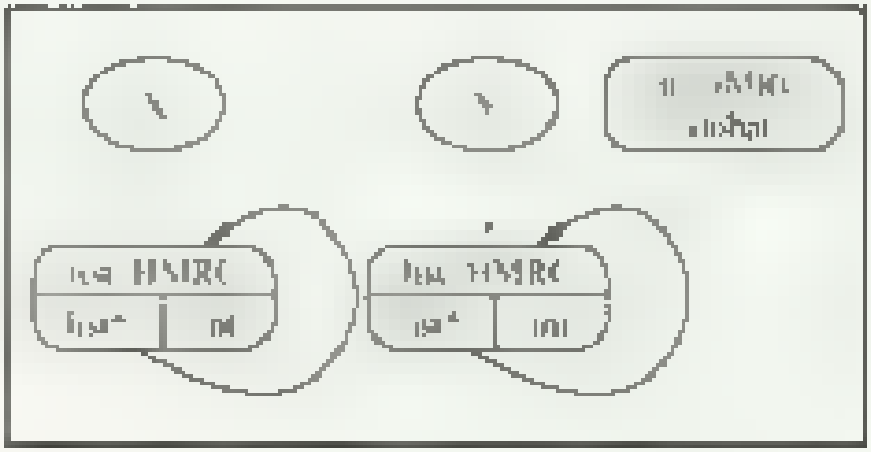
Important aspects of DSA

Field sensitivity + full context sensitivity

Identifies *instances* and *connectivity* of RDS

DSA also captures important properties of memory objects

```
int G;

void twoLists() {
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
}
```



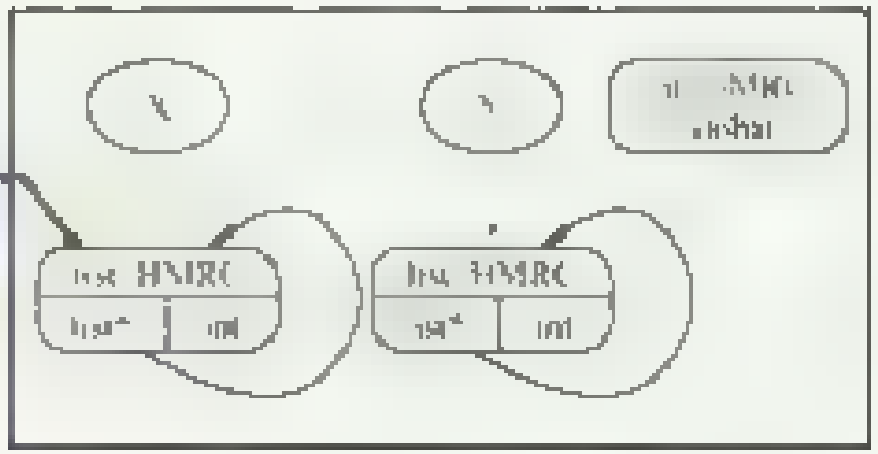
Important aspects of DSA

Field sensitivity + full context sensitivity

Identifies *instances* and *connectivity* of RDS

DSA also captures important properties of memory objects

```
int G;

void twoLists() {
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
}
```



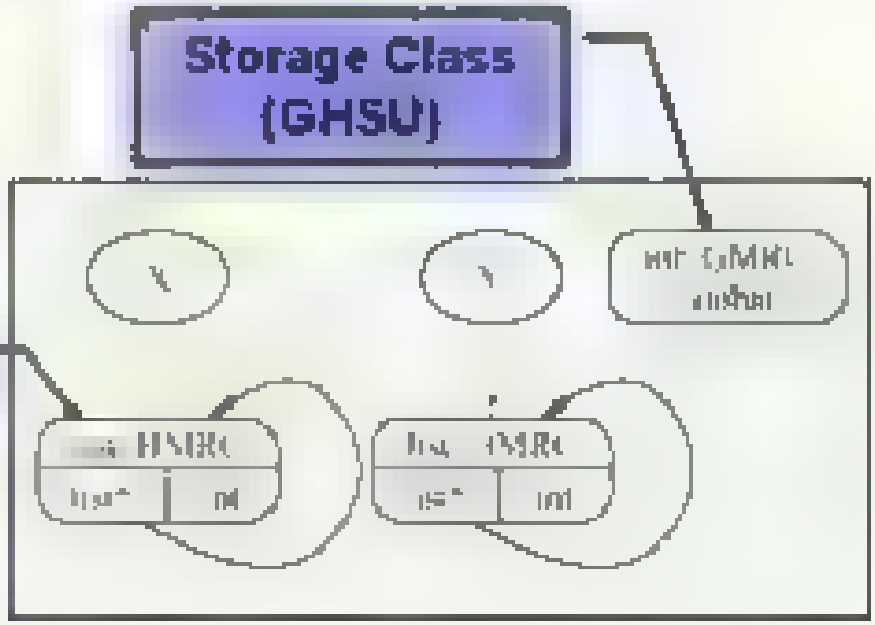
Important aspects of DSA

Field sensitivity + full context sensitivity

Identifies *instances* and *connectivity* of RDS

DSA also captures important properties of memory objects

```
int G;

void twoLists() {
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
}
```



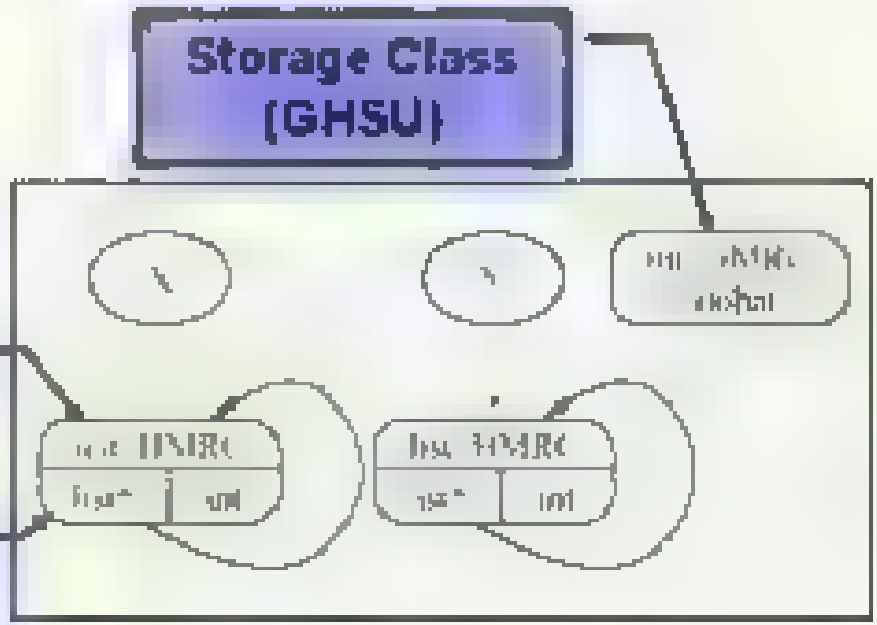
Important aspects of DSA

Field sensitivity + full context sensitivity

Identifies *instances* and *connectivity* of RDS

DSA also captures important properties of memory objects

```
int G;

void twoLists() {
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
}
```



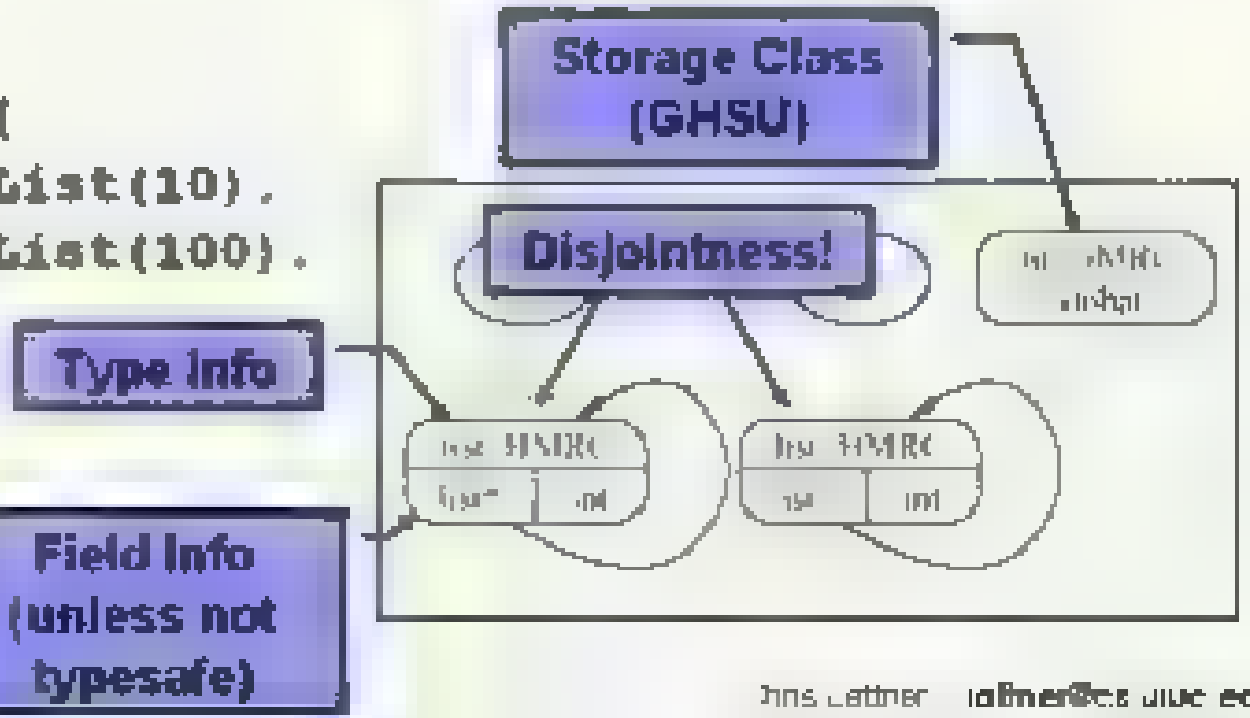
Important aspects of DSA

Field sensitivity + full context sensitivity

Identifies *instances* and *connectivity* of RDS

DSA also captures important properties of memory objects

```
int G;

void twoLists() {
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
}
```



Analysis Time and Memory

Benchmark	#LOC	Time (s)	Mem MB	ΣG (G_{max})
197.parser	11K	0.16 sec	1.06 MB	1506 (60)
larn	15K	0.2 sec	1.1 MB	2740 (49)
186.crafty	21K	0.25 sec	0.96 MB	1996 (107)
moria	36K	0.91 sec	4.7 MB	2433 (76)
255.vortex	67K	2.3 sec	6.1 MB	8597 (85)
254.gap	71K	3.9 sec	12.5 MB	7038 (59)
povray31	137K	7.95 sec	16 MB	26687 (167)

Consistent performance across 35 codes

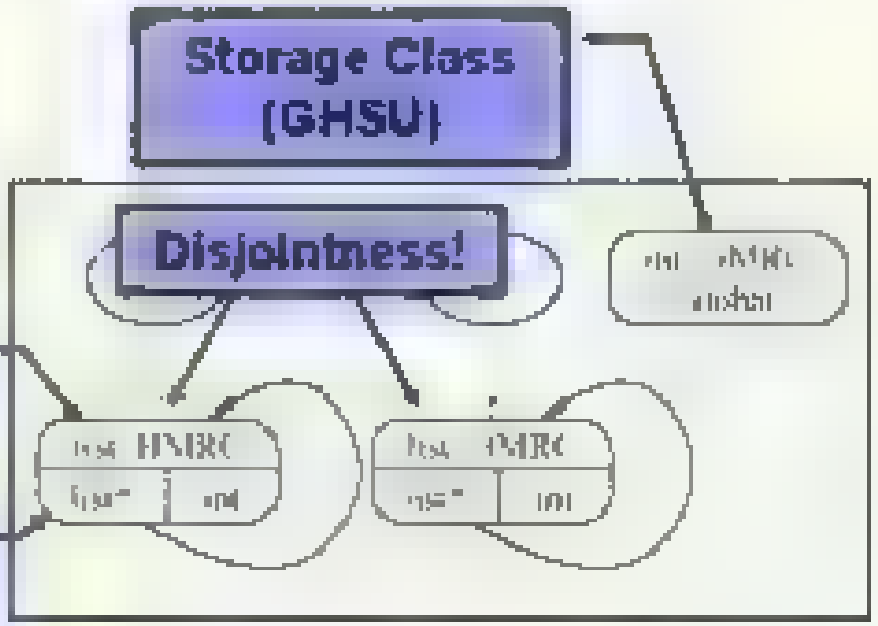
Important aspects of DSA

Field sensitivity + full context sensitivity

Identifies *instances* and *connectivity* of RDS

DSA also captures important properties of memory objects

```
int G;

void twoLists() {
  list *X = makeList(10).
  list *Y = makeList(100).
  addGToList(X).
  addGToList(Y).
  freeList(X).
  freeList(Y);
}
```



Analysis Time and Memory

Benchmark	#LOC	Time (s)	Mem MB	ΣG (G_{max})
197.parser	11K	0.16 sec	1.06 MB	1506 (60)
larn	15K	0.2 sec	1.1 MB	2740 (49)
186.crafty	21K	0.25 sec	0.96 MB	1996 (107)
morla	36K	0.91 sec	4.7 MB	2433 (76)
255.vortex	67K	2.3 sec	6.1 MB	8597 (85)
254.gap	71K	3.9 sec	12.5 MB	7038 (59)
povray31	137K	7.95 sec	16 MB	26687 (167)

Consistent performance across 35 codes

How much is proven type safe?

- **Application of DSA type information:**

  How many loads/stores are

Benchmark	# Typed	# Untyped	Typed %
179.art	372	0	100.0%
181.mcf	371	0	100.0%
164.gzip	1634	61	96.4%

Chris Lattner lattner@cs.uiuc.edu

How much is proven type safe?

- **Application of DSA type information:**
  - How many loads/stores are typed correctly?
- **Type info enables optzn:**
  - e.g. structure reorganization
- **Even for C codes:**
  - Most programs have extensive type information available!
  - Extensive use of custom allocators is the biggest hurdle

Benchmark	# Typed	# Untyped	Typed %
179.art	372	0	100.0%
181.mcf	371	0	100.0%
164.gzip	1654	61	96.4%
186.crafty	8734	383	96.2%
256.bzip2	1911	52	95.1%
175.vpr	4630	371	91.6%
300.twolf	13028	1104	91.6%
183.equake	709	114	87.5%
255.vortex	13397	8015	60.0%
188.ammp	2109	2596	44.8%
197.parser	25747	33179	43.7%
197.parser	1677	2257	41.1%
253.perlbmk	9878	23302	40.9%
254.gap	8433	15117	29.8%
177.mesa	3811	10888	13.5%
Average			68.04%

How much is proven type safe?

- **Application of DSA type information:**
  - How many loads/stores are typed correctly?
- **Type info enables optzn:**
  - e.g. structure reorganization
- **Even for C codes:**
  - Most programs have extensive type information available!
  - Extensive use of custom allocators is the biggest hurdle

Benchmark	# Typed	# Untyped	Typed %
179.art	372	0	100.0%
181.mcf	371	0	100.0%
183.equake	1654	61	96.4%
188.ammp	9734	368	96.2%
256.bzip2	1011	52	95.1%
175.vpr	4939	371	91.6%
300.twolf	13028	1104	91.6%
183.equake	709	114	87.3%
255.vortex	13397	8015	60.0%
188.ammp	2109	2596	44.8%
176.gcc	25747	33179	43.7%
197.parser	1677	2267	41.1%
253.perlbmk	9878	22502	30.5%
254.gap	8432	15117	29.8%
177.mesa	3811	19988	13.5%
Average			68.04%

How much is proven type safe?

- **Application of DSA type information:**
  - How many loads/stores are typed correctly?
- **Type info enables optzn:**
  - e.g. structure reorganization
- **Even for C codes:**
  - Most programs have extensive type information available!
  - Extensive use of custom allocators is the biggest hurdle

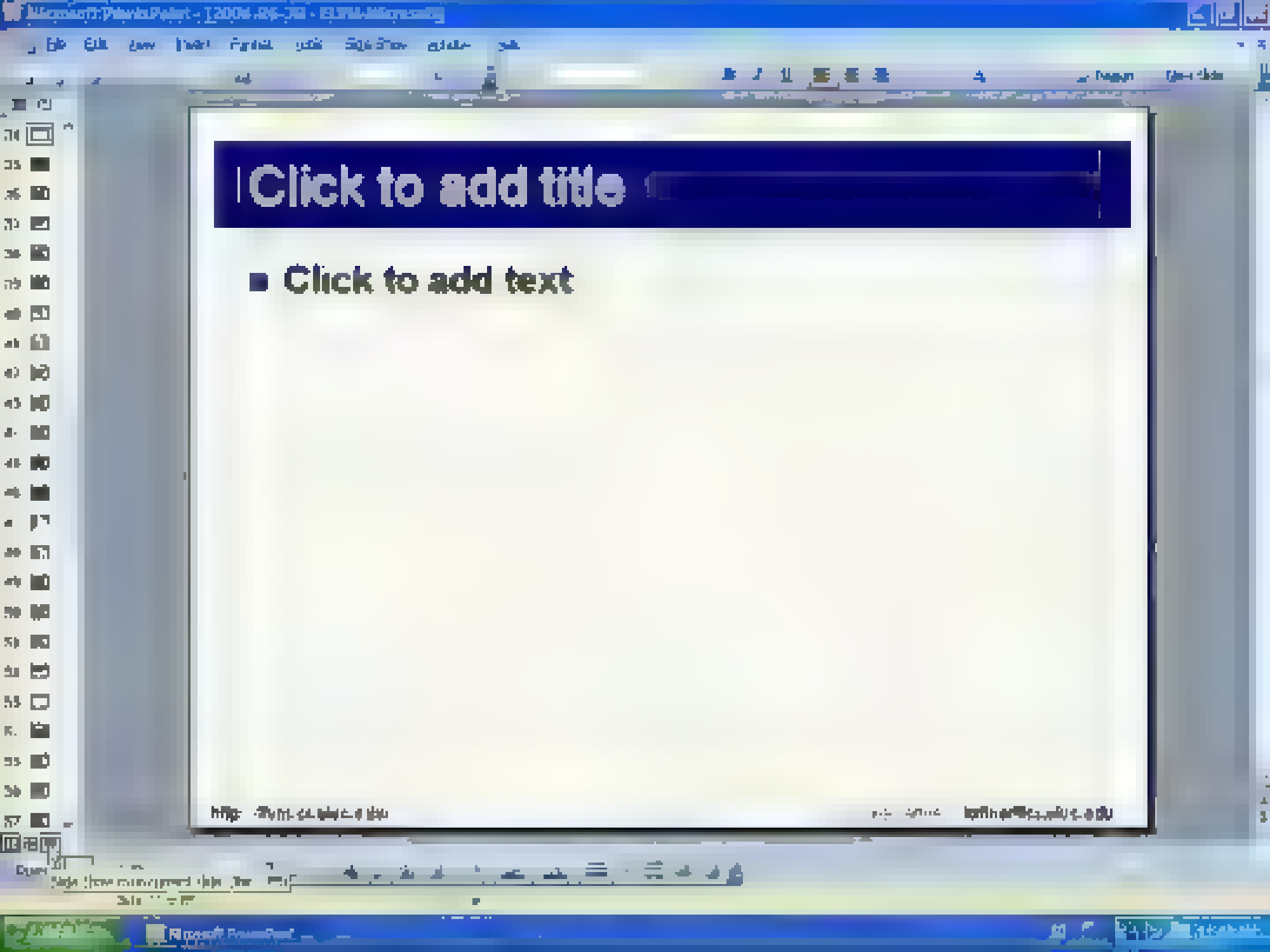
Benchmark	# Typed	# Untyped	Typed %
179.art	572	0	100.0%
181.mcf	371	0	100.0%
183.equake	1654	61	96.4%
188.ammp	9734	353	96.5%
256.bzip2	1011	52	95.1%
175.vpr	4539	371	91.6%
300.twolf	13038	1144	91.6%
183.equake	709	114	87.3%
255.vortex	18397	8015	69.6%
188.ammp	3109	2596	44.5%
186.crafty	25747	33179	43.7%
197.parser	1677	2257	41.1%
253.perlbmk	9678	25303	30.3%
254.gap	8433	15117	29.8%
177.mesa	3911	19088	12.5%
Average			68.04%

Novel Features of DSA

- **Fine-grained tracking of incomplete info:**
  - Memory passed into external function calls
  - Speculative type-safety & field-sensitivity
  - Intermediate analysis results are always correct!
- **Discovers call graph during analysis:**
  - **No iteration** or call graph approximation needed!
- **Field-sensitive for type-safe *memory objects*:**
  - Field-insensitive only in hopeless cases
- **DSA safe for *full generality* of C/C++ codes**
  - and efficient enough to be used!

Novel Features of DSA

- **Fine-grained tracking of incomplete info:**
  - Memory passed into external function calls
  - Speculative type-safety & field-sensitivity
  - Intermediate analysis results are always correct!
- **Discovers call graph during analysis:**
  - **No iteration** or call graph approximation needed!
- **Field-sensitive for type-safe *memory objects*:**
  - Field-insensitive only in hopeless cases
- **DSA safe for *full generality* of C/C++ codes**
  - . and efficient enough to be used!


Click to add title
Click to add text

Novel Features of DSA

- **Fine-grained tracking of incomplete info:**
  - Memory passed into external function calls
  - Speculative type-safety & field-sensitivity
  - Intermediate analysis results are always correct!
- **Discovers call graph during analysis:**
  - **No iteration** or call graph approximation needed!
- **Field-sensitive for type-safe *memory objects*:**
  - Field-insensitive only in hopeless cases
- **DSA *safe* for *full generality* of C/C++ codes**
  - and efficient enough to be used!

Novel Features of DSA

- **Fine-grained tracking of incomplete info:**
  - Memory passed into external function calls
  - Speculative type-safety & field-sensitivity
  - Intermediate analysis results are always correct!
- **Discovers call graph during analysis:**
  - **No iteration** or call graph approximation needed!
- **Field-sensitive for type-safe *memory objects*:**
  - Field-insensitive only in hopeless cases
- **DSA safe for *full generality* of C/C++ codes**
  - and efficient enough to be used!

Talk Outline

- Lifelong Program Analysis & Optimization
- LLVM as a Compiler Infrastructure
- Recursive DS Analysis / Transformations
  - Data structure analysis
  - Automatic pool allocation
- Virtual Instruction Set Computing
- Summary

Pool Allocation

- **Traditional (manual) Pool Allocation:**
  - Custom memory allocators
  - Per-class allocators
  - Often for performance reasons
- **Fully Automatic Pool Allocation:**
  - Each heap node in the DS Graph can become a pool
  - Pools are usually type-homogenous
  - Disjoint data structure instances get separate pools!

Why Segregate Data Structures?

Programs are designed around data structures

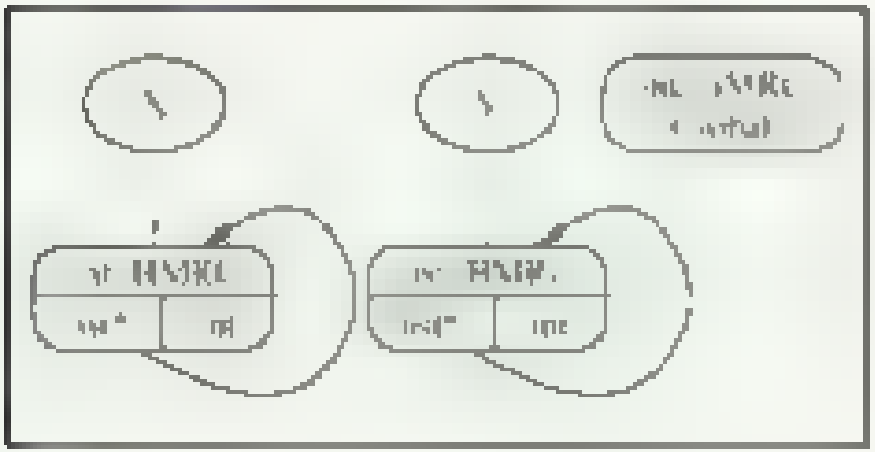
- **Primary Goal: *Better compiler information & control***
  - Compiler knows where each data structure lives in memory
  - Compiler knows order of data in memory (in some cases,
  - Compiler knows type information — runtime points-to graph
  - Compiler knows which pools point to which other pools
- **Incidental benefits: *Better performance***
  - Smaller working sets
  - mproved spatial locality
  - Sometimes convert irregular to regular strides

Automatic Pool Allocation Overview

- Each DS node instance uses separate pool (for now)
- Each pool can be type-homogeneous
- Retain explicit `free()` for objects

Automatic Pool Allocation Overview

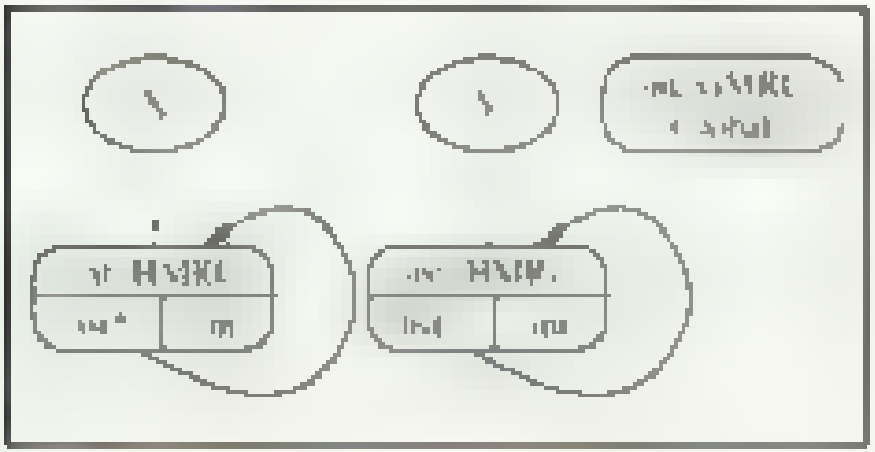
- Each DS node instance uses separate pool (for now)
- Each pool can be type-homogeneous
- Retain explicit `free()` for objects

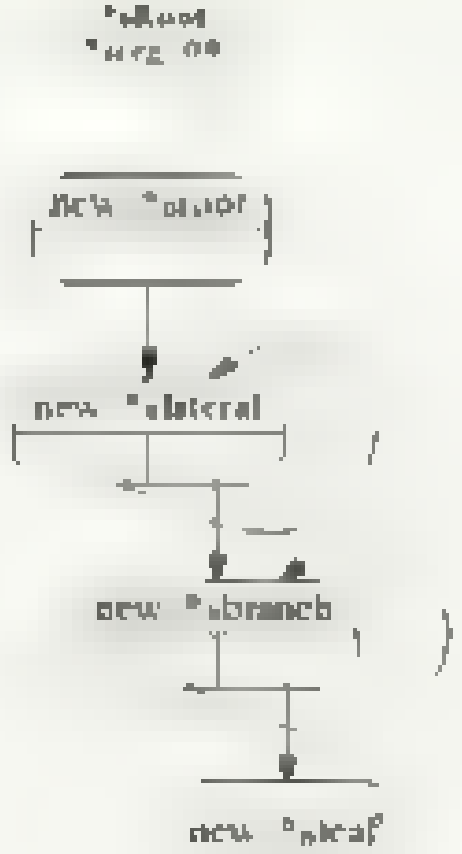


Chris Lattner lattner@cs.uiuc.edu

Automatic Pool Allocation Overview

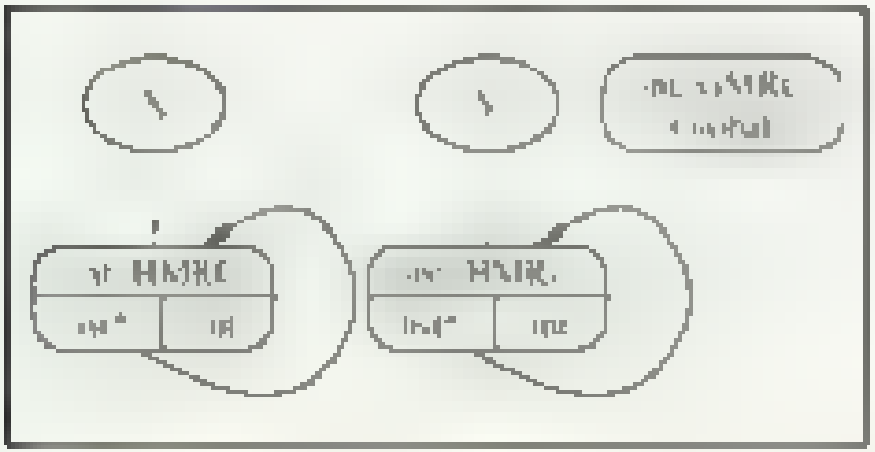
- Each DS node instance uses separate pool (for now)
- Each pool can be type-homogeneous
- Retain explicit `free()` for objects

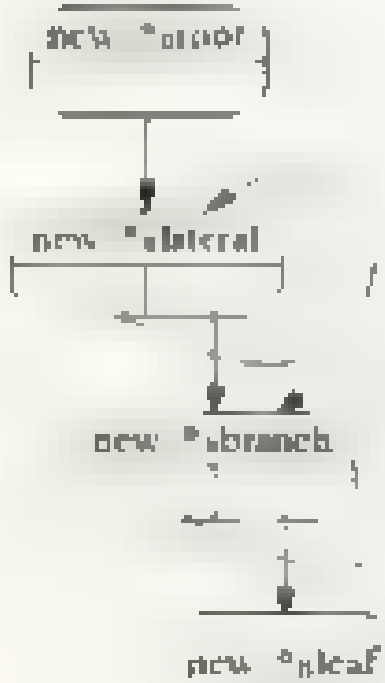




Automatic Pool Allocation Overview

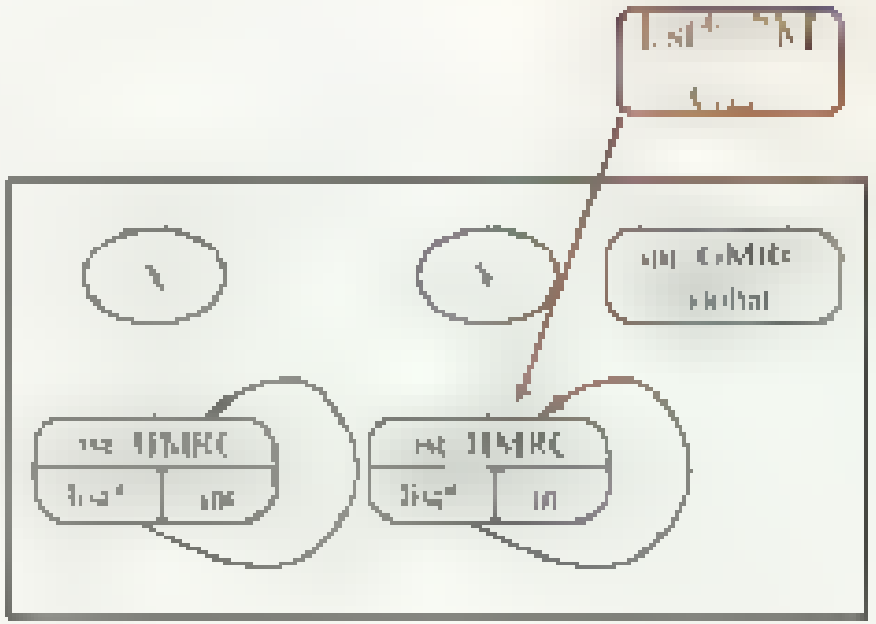
- Each DS node instance uses separate pool (for now)
- Each pool can be type-homogeneous
- Retain explicit `free()` for objects





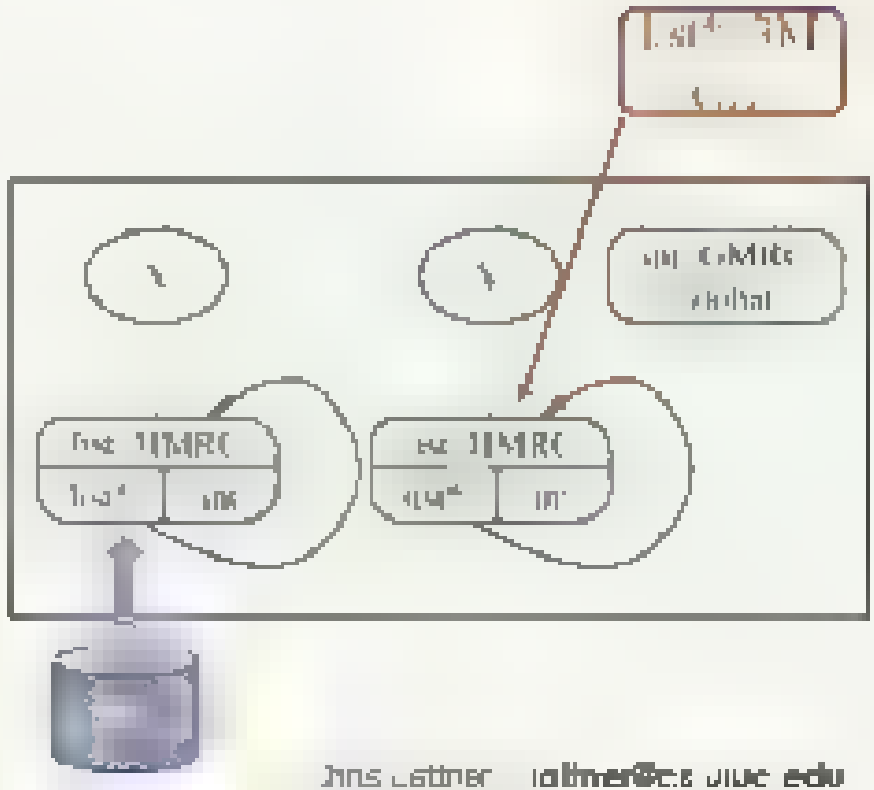
Pool Allocation: Example

```
...t *makeList(...
  ...t *New = malloc(...
  New->Next = ...  makeList(...
  New->Data = ...; return New;
}

...t twoLists(
  ...t *X = makeList(...
  ...t *Y = makeList(...
  GL = Y;
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
```



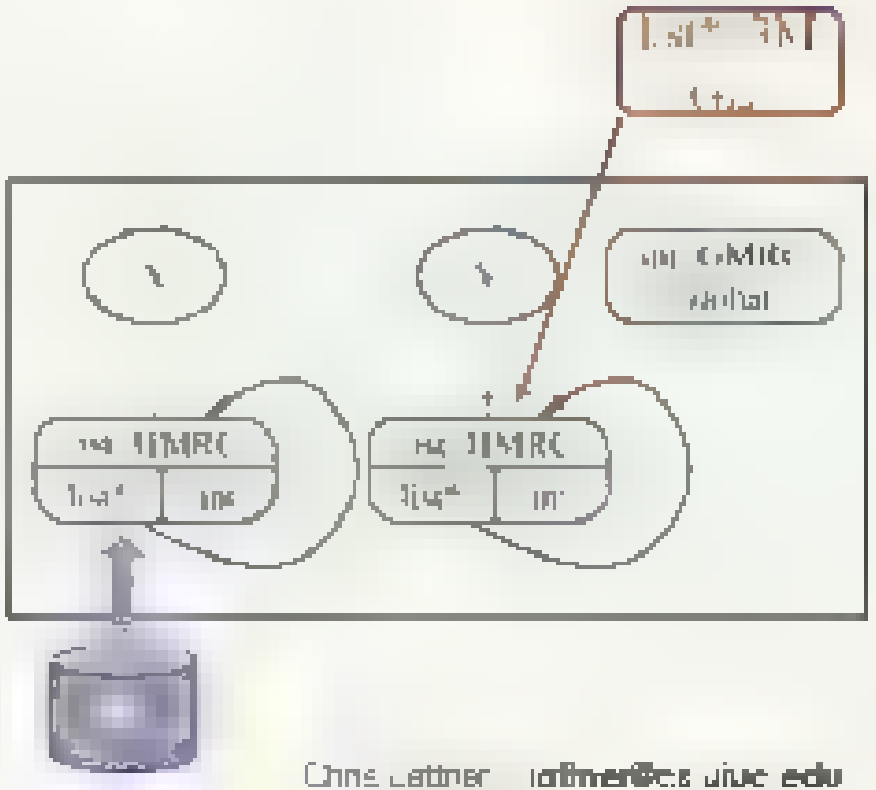
Pool Allocation: Example

```
List *makeList(int N) {
  List *New = malloc(sizeof(List));
  New->Next = N > 0 ? makeList(N-1) : 0;
  New->Data = N; return New;
}

int twoLists() {
  List *X = makeList(N);
  List *Y = makeList(N);
  GL = X
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
}
```



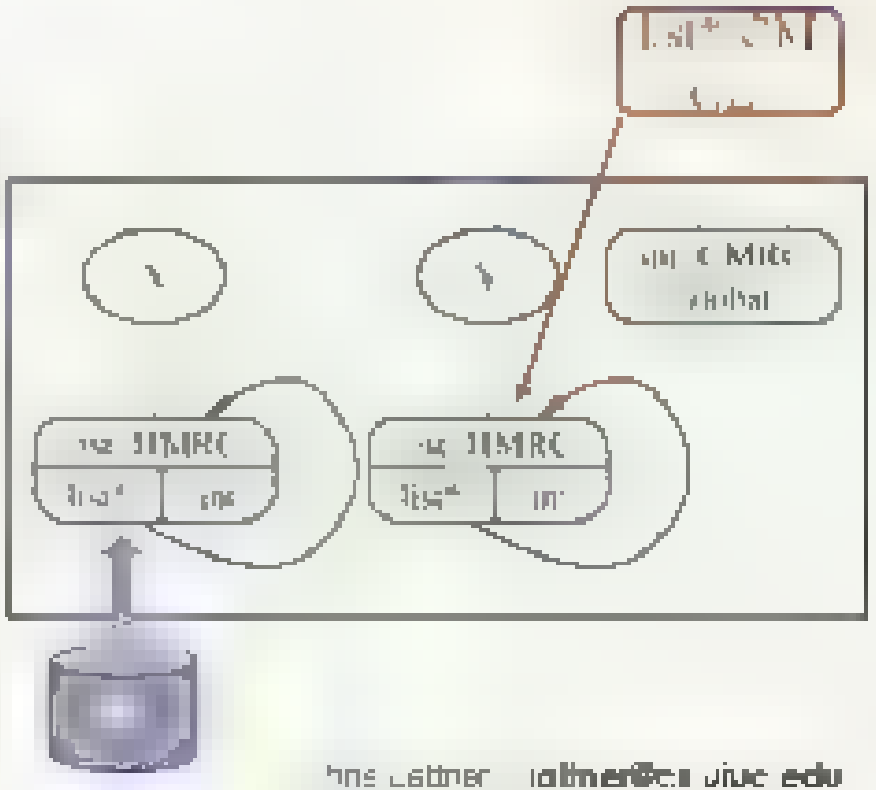
Pool Allocation: Example

```
List *makeList(int Num,
  List *New = malloc(sizeof(List));
  New->Next = Num ? makeList(Num-1) : 0;
  New->Data = Num; return New;
}

int twoLists() {

  List *X = makeList(1000);
  List *Y = makeList(1000);
  GL = Y;
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
```



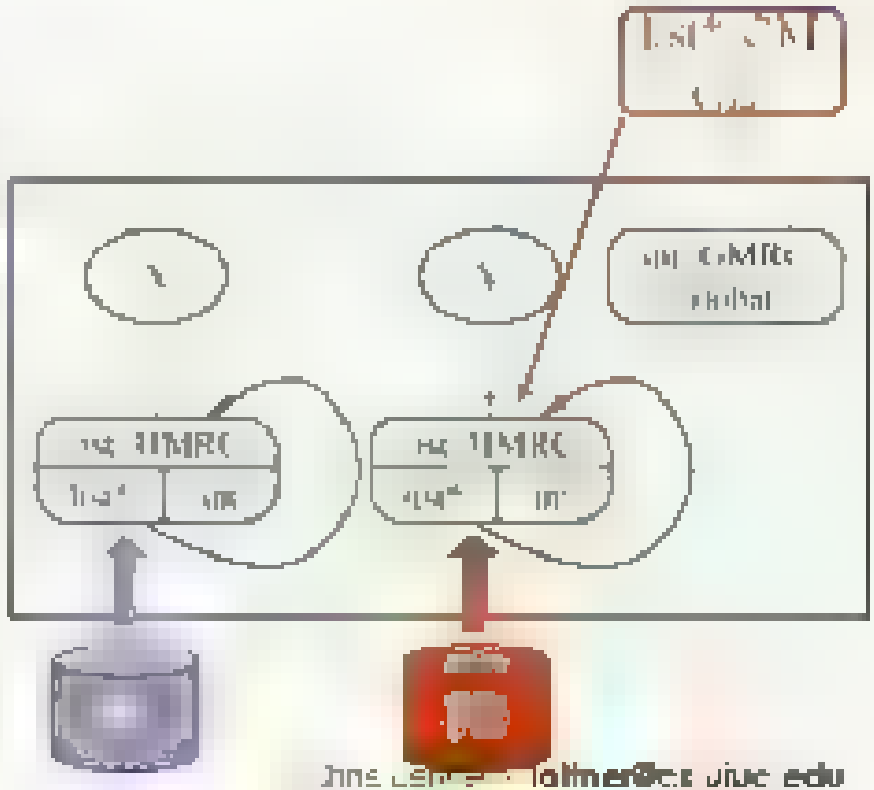
Pool Allocation: Example

```
...* *makeList(...
  ... *New = ...
  New->Next = ..., makeList(... - ...) ...;
  New->Data = ...; ... New;

... twoLists(...
  ...
  ... *X = makeList(1 ...);
  ... *Y = makeList(... );
  GL = Y
  addGToList ...);
  addGToList ...);
  freeList ...
  freeList(Y);
  ...
```



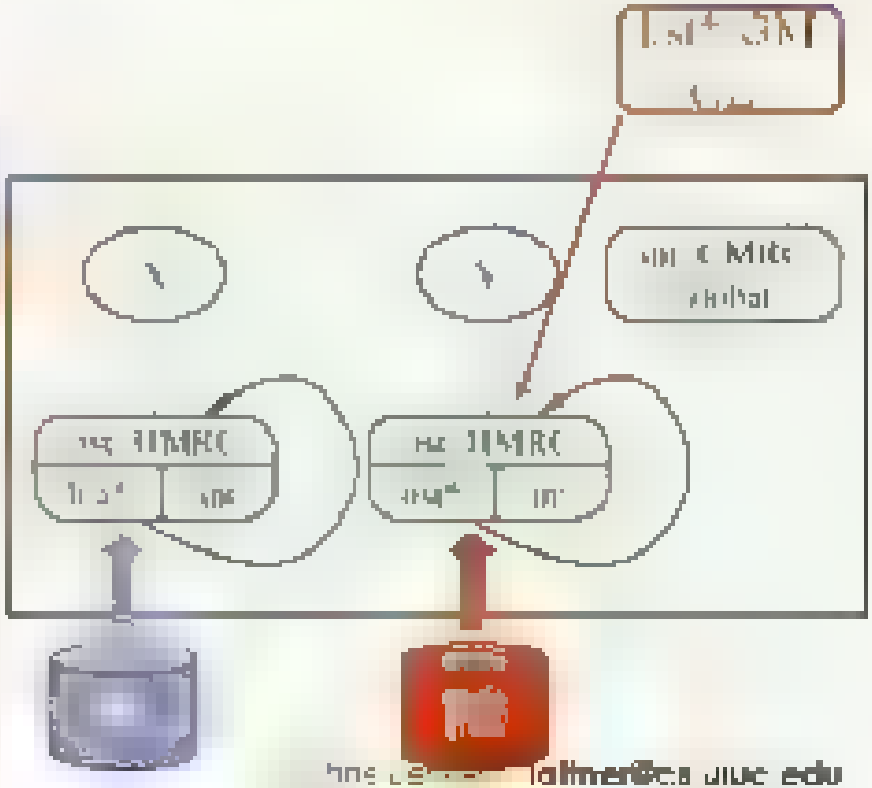
Pool Allocation: Example

```
...t *makeList(...
  ...t *New =
  New->Next = ..., makeList(...
  New->Data = ...;
}

...t twoLists(
  {
  ...t *X = makeList(1
  ...t *Y = makeList(...);
  GL = X;
  addGToList ...);
  addGToList ...);
  freeList ...
  freeList(Y);
}
```



Pool Allocation: Example

```
List *makeList(int
  List *New =
  New->Next = ...makeList(...-1...) : 0;
  New->Data = ...; return New;

int twoLists
  ...
  List *X = makeList(1
  List *Y = makeList(
  GL = Y;
  addGToList(X);
  addGToList(...);
  freeList
  freeList
```



High-level pool allocation algorithm

1. Use DSA to identify data structures on the heap
   - Identifies distinct instances and type information
2. Determine lifetime of data structures
   - Escape analysis for entire data structures
3. Create pools, add pool arguments to functions
   - Interprocedural code restructuring
4. Rewrite function bodies to call poolalloc/poolfree instead of malloc/free

What about global variables?

- **The problem:**
  - Function accesses a node reachable from a global
  - Node escapes all functions except main

    ⇒ Must pass pool descriptor all the way from main()
- **Solution:**
  - Make pool descriptor a global variable itself
  - Initialize it in main, access it directly where needed
  - **Greatly** reduces # pool args in some programs

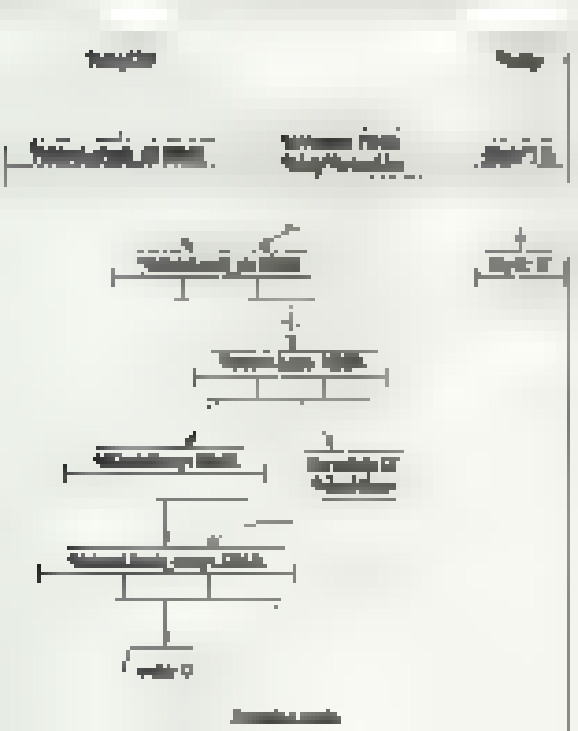
Two more optimizations

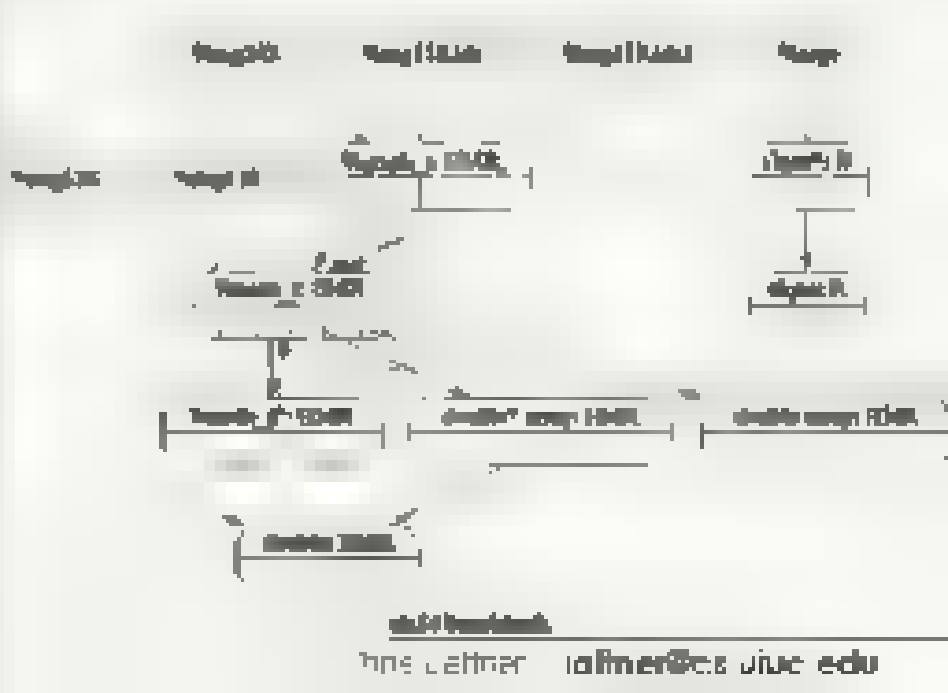
- **Intelligent placement of poolinit/poolfree:**
  - Init as late as possible, destroy as early as possible
  - Standard dataflow analysis
  - Can be extended interprocedurally *[Aiken et al., PLDI 96]*
- **Elide poolfree calls:**
  - 'pooldestroy' releases all pool memory to the system
  - No need to call poolfree right before a pooldestroy
  - Can make entire **traversals** dead (ADCE removes loop).

```
for (list *L = ..., L; L = L->Next) poolfree(PD, L);
pooldestroy(PD);
```

Pool Coalescing for Locality

- **So far, we've assigned each node to a pool:**
  - Keeps nodes homogenous
  - Better support for later analyses & transformations
- **Real data structures have multiple pools:**
  - Pool allocate subset? Make some share a pool?





Program pool properties

- Large programs have many pools
- # args added is very reasonable
- Most pools are type-safe

Program	LOC	Static Pools	Type Safe	Dyn. Pools	Num Args
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Program pool properties

- **Large programs have many pools**
- **# args added is very reasonable**
- **Most pools are type-safe**

Program	LOC	Static Pools	Type Safe	Dyn. Pools	Num Args

Pool alloc performance effect

- **2/3 programs: no substantial gain/loss**

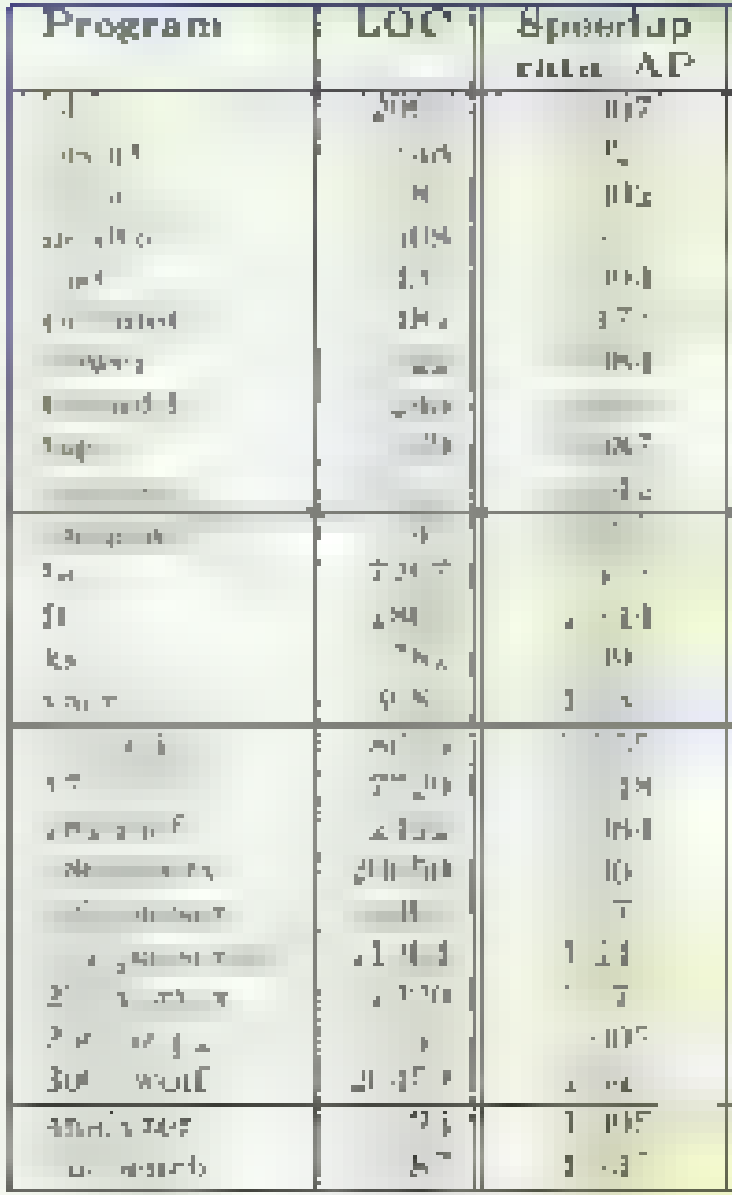
1 case: 9.4% gain

4 cases: ~20% gain

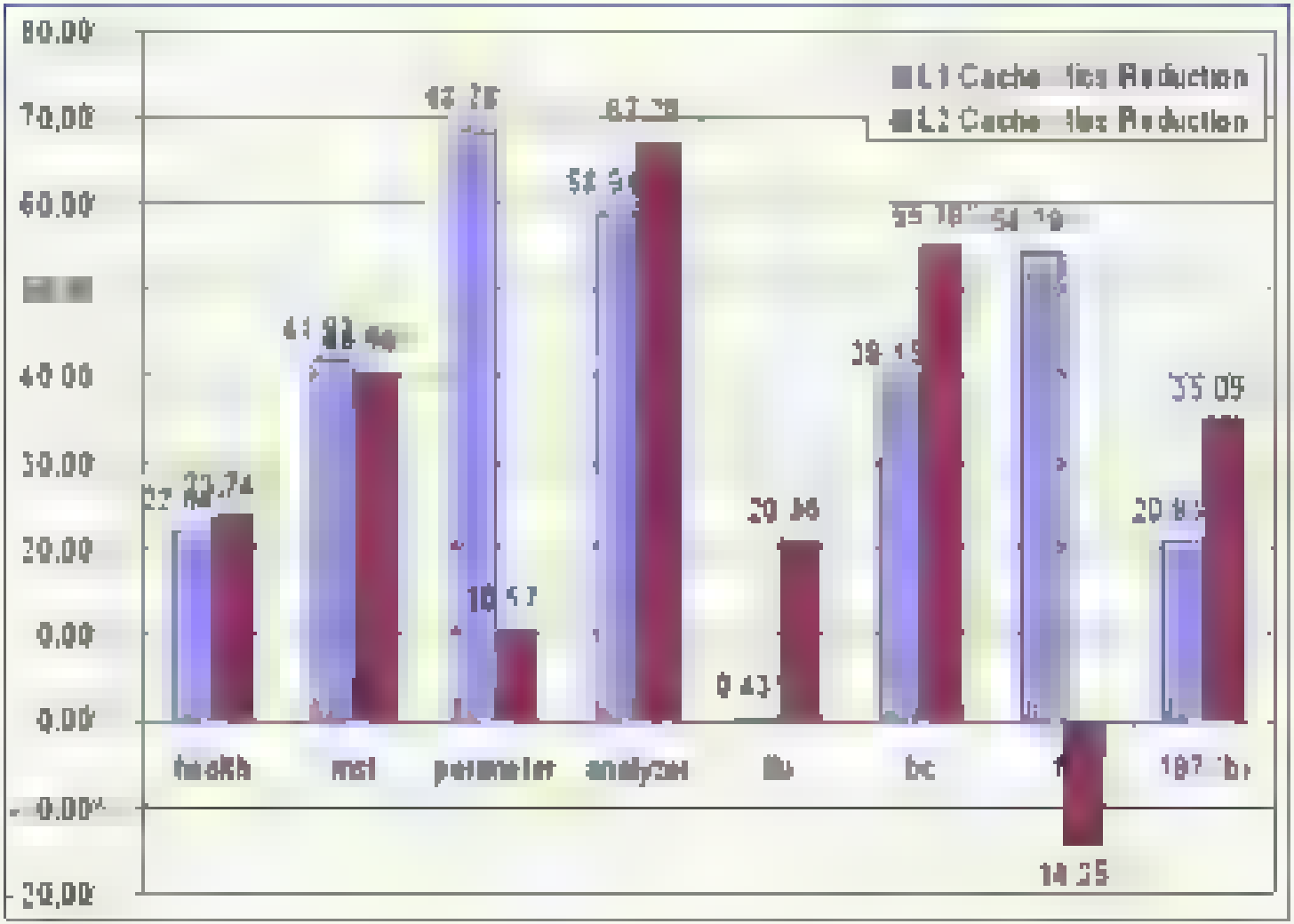
2 cases: 34/41% gain

1 case: 100% gain

- **Unlike applications, benchmarks typically don't fragment heap!**



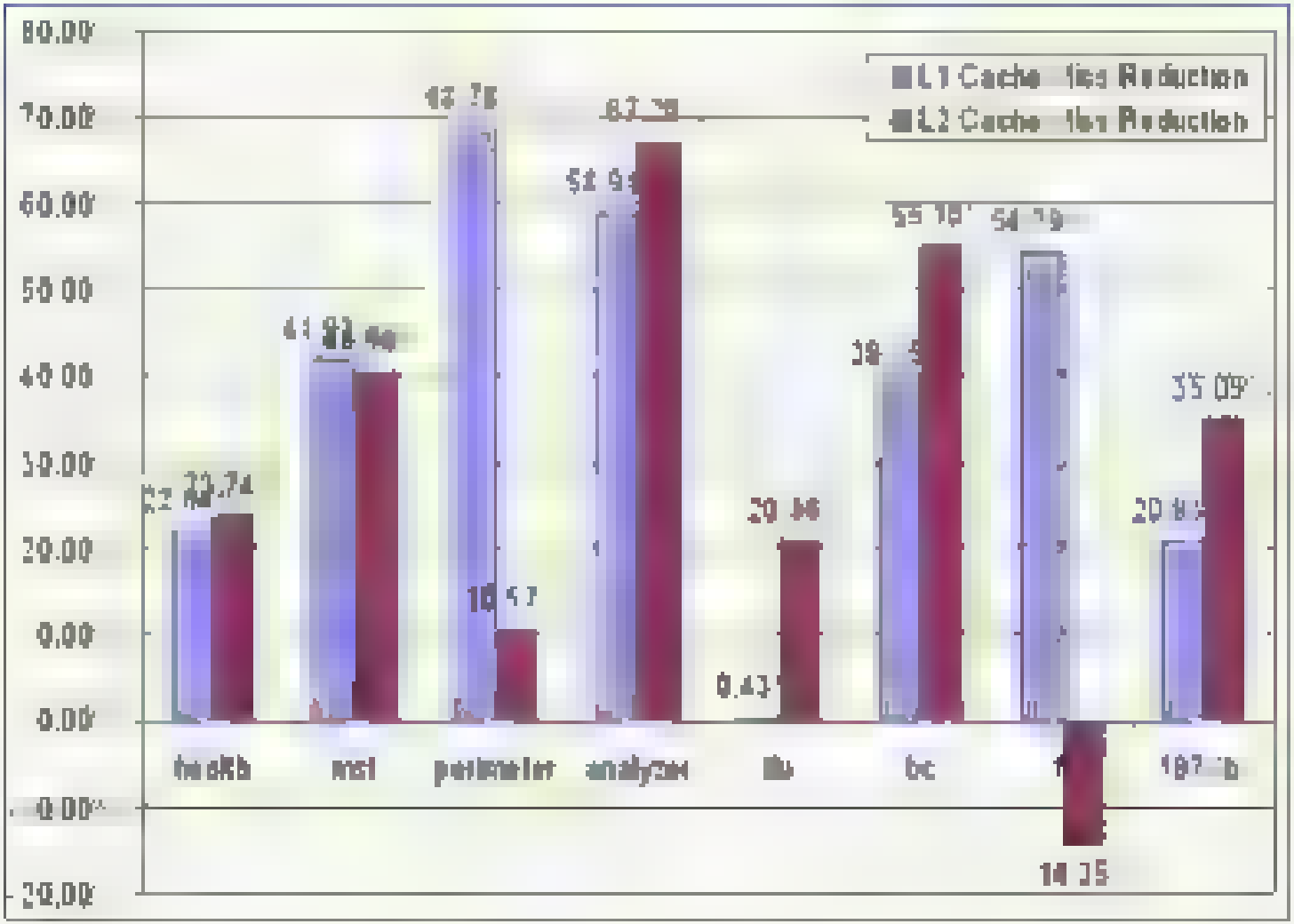
Cache Miss Reduction



Miss rate measured with perfctr on 3Ghz Pentium 4 Xeon

Chris Lattner lattner@cs.uiuc.edu

Cache Miss Reduction



Miss rate measured with perfctr on 3Ghz Pentium 4 Xeon

Chris Lattner lattner@cs.uiuc.edu

Pool alloc performance effect

- **2/3 programs: no substantial gain/loss**

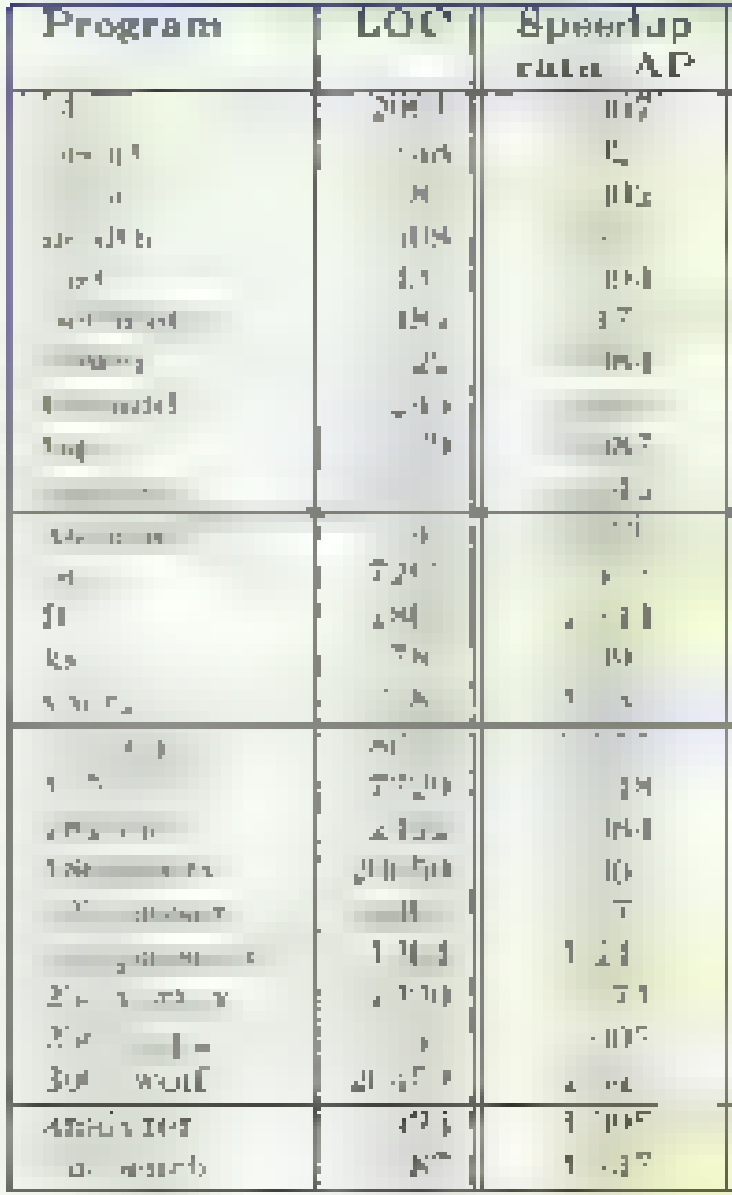
1 case: 9.4% gain

4 cases: ~20% gain

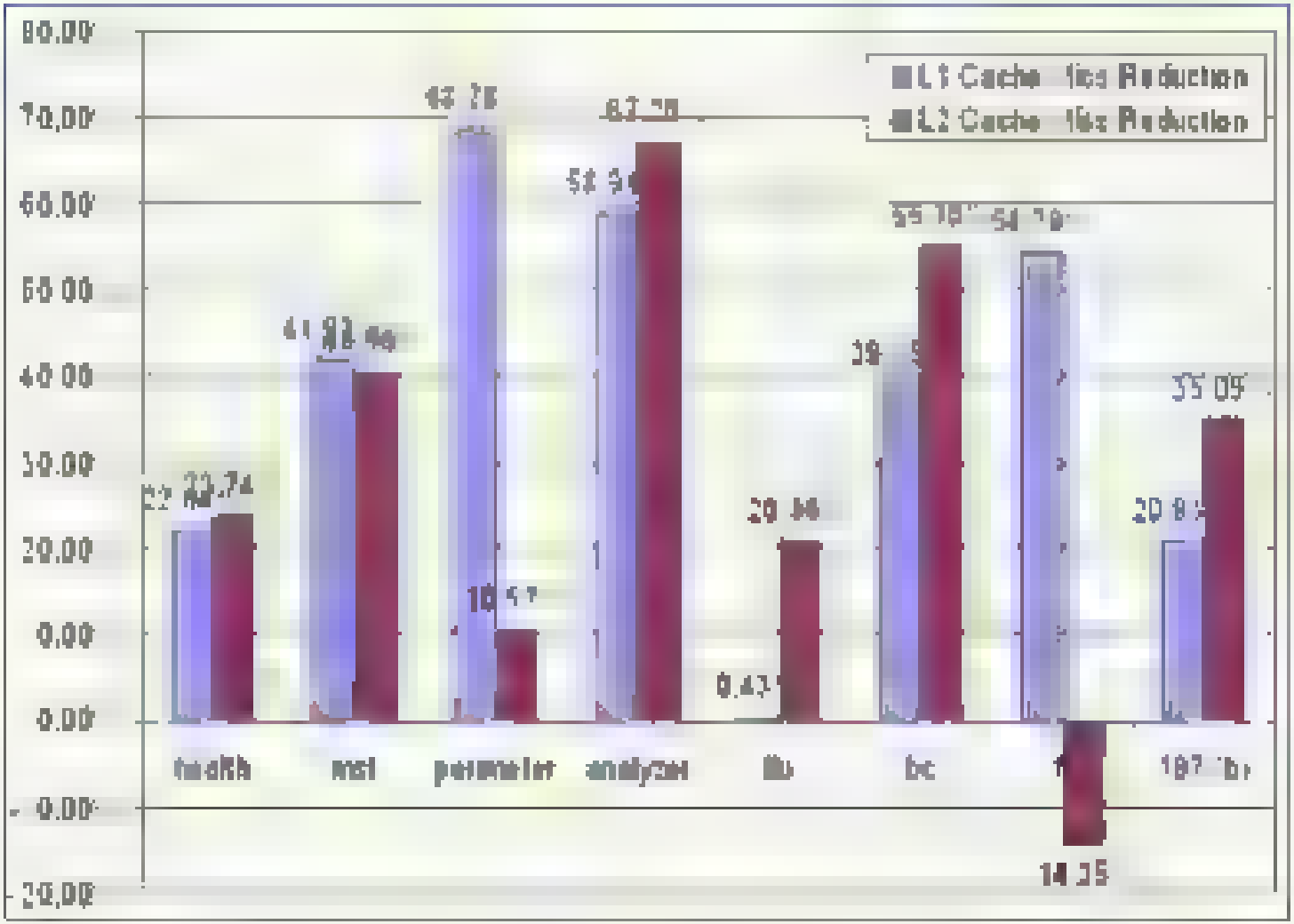
2 cases: 34/41% gain

1 case: 100% gain

- **Unlike applications, benchmarks typically don't fragment heap!**



Cache Miss Reduction



Miss rate measured with perfctr on 3Ghz Pentium 4 Xeon

Chris Lattner lattner@cs.uiuc.edu

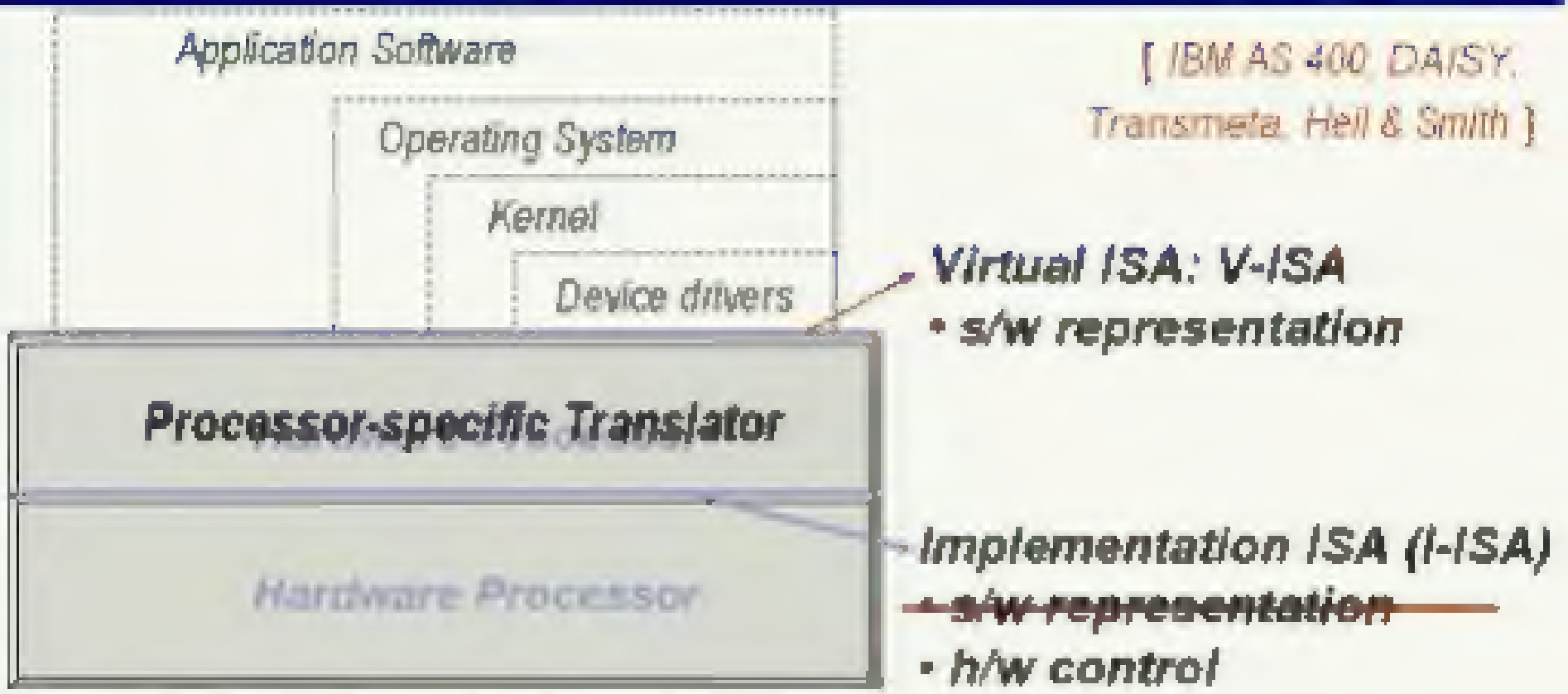
Pool Allocation Highlights

- **Transform data structures to use pool library**
- **Give compiler info & partial layout control**
- **Can dramatically improve DS locality:**
  - Defragment data structures
  - Often lays out nodes in allocation order
- **Opens the door for new applications!**
  - by combining static analysis with runtime control

Talk Outline

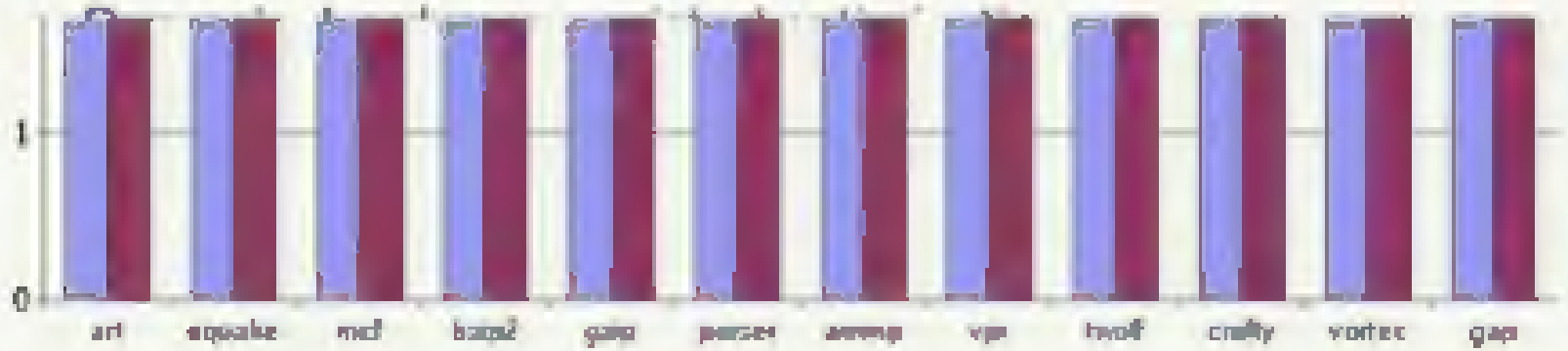
- Lifelong Program Analysis & Optimization
- LLVM as a Compiler Infrastructure
- Recursive DS Analysis / Transformations
- Virtual Instruction Set Computing
- Summary

VISC: *Virtual* Instruction Set Computers



Future VISC Research Goals

- **Processor and Micro-architecture Design**
  - New software-controlled micro-architectural mechanisms
  - Explicitly parallel micro-architectures, visible only to translator
  - Hardware fault tolerance through software translation
- **Compilers and Optimization**
  - High-level abstractions of parallelism in the V-ISA



Average for x86: About 2.6 instructions per LLVA instruction

Average for Sparc: About 3.2 instructions per LLVA instruction

⇒ *Very small semantic gap ; clear performance relation*

Summary

- **LLVM Compiler Infrastructure**
  - Low-Level IR, High-Level capabilities
  - Strong platform for supporting research
  - Publicly available: http://llvm.cs.uiuc.edu/
- **Macroscopic Data Structure Techniques**
  - Analyze and transform entire data structures
  - Works on real programs (hopefully soon MSIL too!)
- **VISC Processor Design**
  - Give architects ability to innovate with their ISA
  - Many hard and interesting problems remain

Any (more) questions??